COSC427: Advanced OO Design

Exam 2008

Time allowed:	**2 hours**.
Number of pages:	**7**.
Number of questions:	**6**.
Total marks:	**100**.
Resources:	**Open book**: You may use any printed or electronic resources. You may not communicate, directly or electronically, with other people. If you want to print anything, ask a supervisor to collect it from the printer for you.
Answers:	Write all answers in your **answer book**.

1. **[40 marks]** This question refers to the UML class diagram of the *Stockbrokers design* in Figure 1 and explanatory notes in Figure 2. Criticise the design. For each criticism, cite any maxims, principles, smells, etc that back up your arguments, and clearly explain how you would improve the design to address the problem. Use UML where it aids communication. Document any assumptions you need to make.

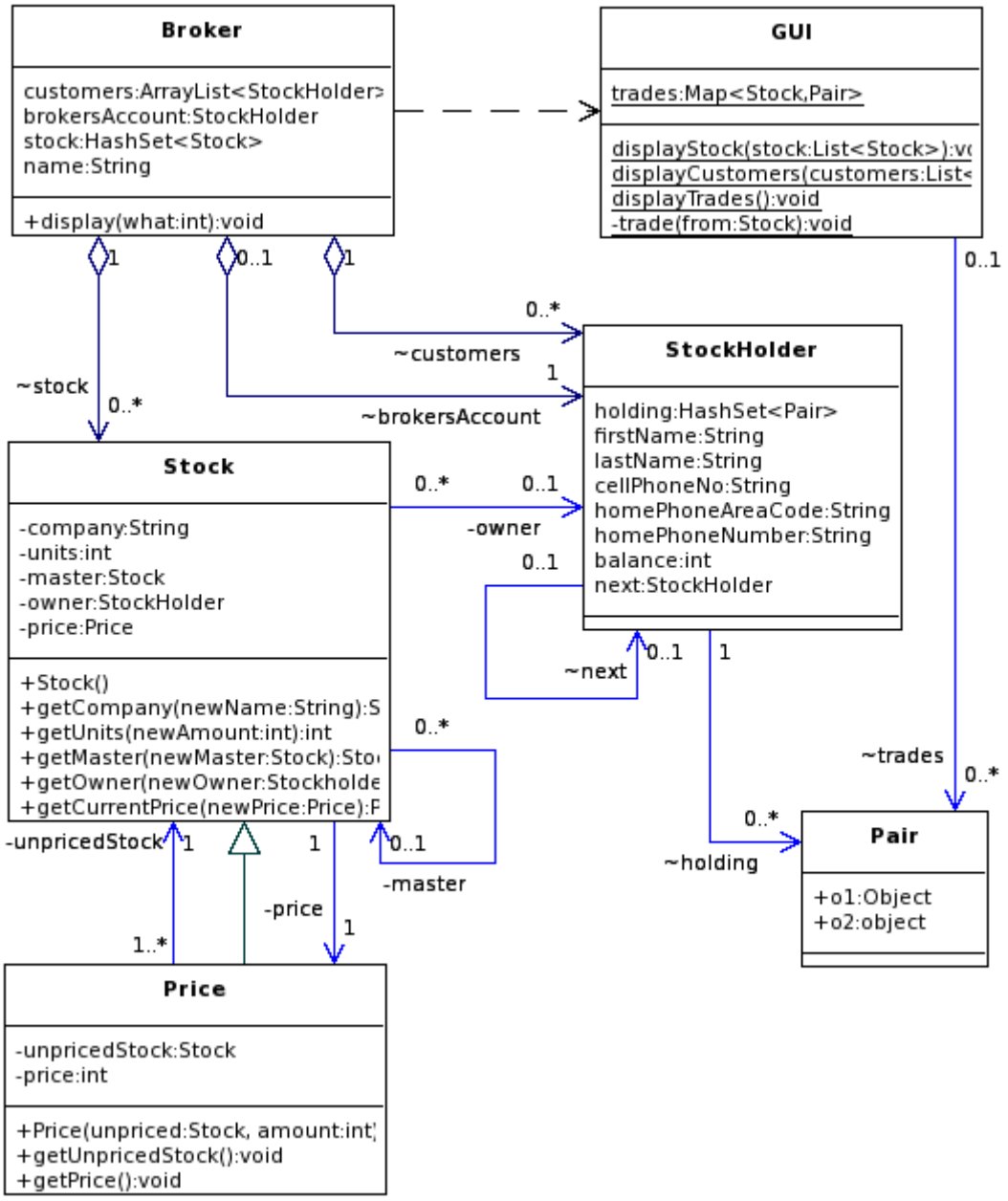

Figure 1: StockBrokers system class diagram

- This design keeps track of a stockbroker's customers, current stock prices, and trades of stock between customers.
- `Broker` represents a stockbroker. It stores a list of `StockHolders`: the customers who trade stock through this `Broker`. A `Broker` also has a `brokersAccount`, which is another `StockHolder` instance that keeps track of contact and financial info about the `Broker` itself rather than about a customer. `Broker` also stores a collection of all `Stock` master records (see below for more on master records).
- `Broker` has a `display()` method that accepts a flag telling it what to display. This method then forwards requests to `GUI` methods.
- `GUI` can display various screens of info, and carry out stock trades requested by the user. All of the methods of `GUI` are static.
- `GUI` also stores a record of all past `trades`. It does this using a `Map` indexed by a master `Stock` record (the seller's holding), with a value which is a `Pair` object. The `Pair` records the `Stock` object resulting from the trade (the buyer's holding) and the date of the trade (in `o1` and `o2`, respectively).
- `StockHolder` stores `name`, `phone` and `balance` for a customer account. If a customer has more than one account, the `next` reference points to another `StockHolder` instance, which represents an additional account for that customer. That too may have a `next` pointer for a 3rd account, and so on.
- `StockHolder` also stores a set of `Pair` objects to keep track of `Stock` owned by that `StockHolder`. If a `StockHolder` currently owns a `Stock`, the `Price` object for that `Stock` is stored in `o1`, and the date of purchase in `o2`. If the `Stock` has already been sold by the `StockHolder`, this is indicated by reversing the 2 fields of `Pair` so that `o1` holds the sale date and `o2` holds the `Price` object.
- `Broker`, `GUI` and `StockHolder` are in a package together. The fields of these 3 classes have package access, so they can access each other's data without exposing it to any classes from other packages.
- `Stock` represents the sellable shares of a company. It stores the company `name`. `Stock`'s methods allow its fields to be read and changed; each method returns a field's value, and if the parameter is non-null, the field is also updated to match the parameter.
- If a `Stock`'s `master` field is null, that means the `Stock` object is a master record, which represents all the shares for a company. In this case `units` contains the total number of shares in circulation, and owner must be null. The `price` is the current sharemarket purchase price.
- If a `Stock`'s master field is not null (i.e. it references a `Stock` master record), this object records ownership of some shares of the `Stock`. The `owner` is the StockHolder, and `units` indicates how many shares are owned. The `price` field stores the price paid at the time of purchase.
- `Price` records how much a share of a `Stock` cost at some point in time. Whenever a price changes, a new `Price` object is constructed, wrapping up the `Stock` to which it applies.
- `Pair` is a utility class that can hold any two `Objects`.

Figure 2: Stockbrokers system notes

2. **[14 marks for whole question]** For each statement in parts (a) – (l) provide a word or phrase that *best* captures the idea (give a wiki page name if possible) and write a concise sentence explaining your answer.

   (a) [2 marks] A *maxim* that contrasts with the culture of software reuse.

   (b) [2 marks] A *maxim* that might be measured automatically by software tools.

   (c) [2 marks] A *maxim* that can never be measured automatically by software tools.

   (d) [2 marks] A *maxim* violated by the GoF Strategy pattern.

   (e) [2 marks] A *GoF pattern* that allows a single object to do the job of multiple similar objects by externalising their changeable properties.

   (f) [2 marks] A *pattern* that encourages the use of abstract getters and setters instead of accessing attributes**.**

   (g) [2 marks] A *pattern* likely to be most helpful when developing a system to record the contents of a warehouse and the associated income and expenditure.

3. **[4 marks]** A well known quote from Kent Beck says: "Make it work. Make it right. Make it fast." What does he mean, and why?

4. **[4 marks]** What is the principal message of Foote and Yoder's *Big Ball of Mud* paper? Write as concise a synopsis as possible.

5. **[6 marks]** Cohesion and coupling are influential software design concepts. Identify maxims (as many as you can) that encourage high cohesion and low coupling. Explain your answers.

6. [32 marks for whole question] The following questions refer to the UML class diagram in Figure 3 and explanatory notes in Figure 4. Some getters and setters and other details are omitted from the diagram, but may be assumed where necessary. (Recall that a # in UML means protected, and an underline means static.) Document any non-trivial assumptions you need to make.

(a) [24 marks] Find as many Gang of Four design patterns as you can in the Irrigation design. Name each pattern and describe where and how it is used in this design. Provide just enough information to make it clear how and why the pattern is applied here. Note any important variations from the standard pattern. There is no need to comment on the value of the pattern.

(b) [8 marks] How would you extend this design to allow irrigation system models to be stored in (and retrieved from) a relational database? Explain your design, including any additional patterns used. Use UML where it aids communication.

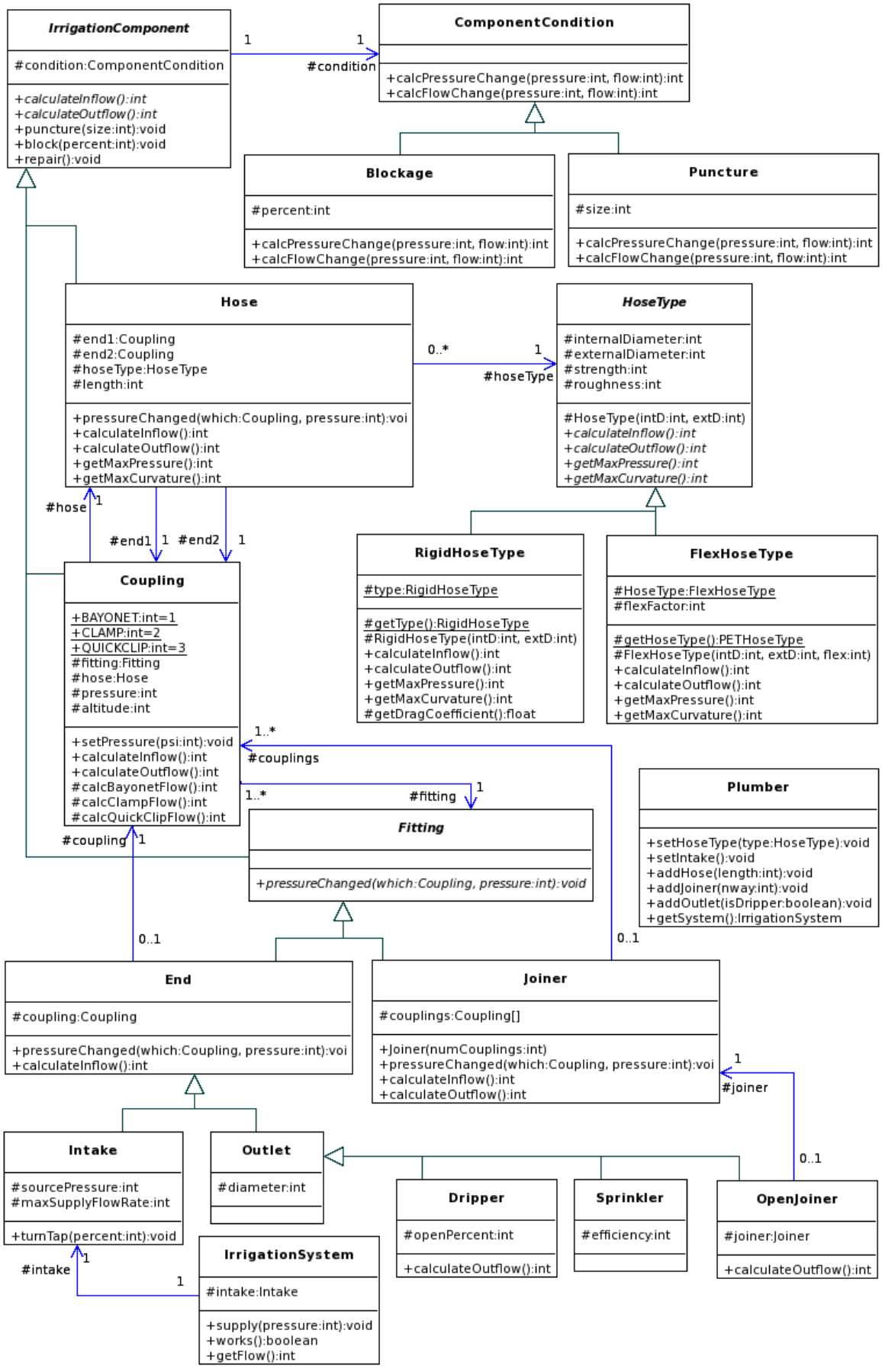

Figure 3: Irrigation system class diagram

- This software models assembly and performance of irrigation systems.
- A system is assembled from Hoses and Fittings, joined by Couplings.
- A Hose is a length of tube, of a particular HoseType. Most of Hose's methods are implemented by simply forwarding requests to HoseType, passing Hose length when necessary.
- A HoseType represents an available hose product, such as 13mm PVC pipe. Currently only two such products are supported, but more may be added later. The constructors of HoseType and its subclass are protected to ensure no client class constructs instances. Hoses in existing systems don't usually change their HoseTypes, but it is helpful to be able to model systems constructed from different materials.
- A Fitting plugs in to 1 or more Hoses (via Couplings), depending on what kind of fitting it is.
- A Joiner is a Fitting that has 2 or more Couplings, so it connects Hoses.
- An End attaches to 1 hose and so is a terminal Fitting. An Intake supplies water to the system, while an Outlet emits it.
- An OpenJoiner acts like an outlet, but is in fact a Joiner with an unconnected coupling.
- A Coupling joins a Fitting to a Hose. The flow calculations depend on the kind of coupling (BAYONET, etc).
- Each IrrigationComponent has a condition that can change over time. A component that is working normally uses a direct instance of ComponentCondition (whose methods leave pressure and flow unchanged), but a blockage or puncture in the component can be modelled using a Blockage or Puncture condition.
- A Plumber makes assembly of systems easier. Its methods allow IrrigationComponents to be created and automatically joined to the last unconnected Coupling of the right type. The getSystem() method provides access to the resulting system.
- IrrigationSystem simplifies use of an irrigation model, for clients that need only to test its performance in straightforward ways. More sophisticated clients can acces the IrrigationComponents directly.

Figure 4: Irrigation system notes

END OF PAPER